1. ABSTRACT
In this contribution we consider a light weight approach to support software architecture design over the web. We outline the architecture description language we use for this purpose which is based on the concept of a self contained software component. Based on this concept a loosely coupled approach to combine remote repositories is supported. A number of analysis tools are sketched which are available as back end services within this approach. These tools not only allow to check syntactic and semantic properties but also allow to assess a given architecture wrt. quantitative properties like performance.

1.1 Keywords
Software architecture, WWW tool support, performance evaluation, dynamic semantic-directed system configuration

2. INTRODUCTION AND RELATED WORK
Designing the software architecture of a distributed software system has been relieved from tedious tasks like designing subsystems for communication and / or persistence services. However, designing the overall system structure of usual complexity is by no means easy. In addition, the task to set up an initial architecture and maintain and develop it further collaboratively requires certain concepts for representing and checking architectural properties on the one side and appropriate collaboration mechanisms on the other side.

The aim is to have a number of cooperating light weight tools which allow to design a software architecture over the web. There are, of course, some more complex analysis tools necessary which are implemented as servers which are publicly available. The basic idea is that the various developers publish their related repositories over the web forming a large common pool of software components and possibly partial architecture descriptions. Services like analysis are then invoked on a server while the architecture design and composition process is done at the designer’s site.

Due to the low degree of control over the state of component descriptions in remote repositories a self contained component model of software is desirable. This means that the description of a component does not contain references to other components, just descriptions of requirements to other components are included. Such an approach is the architecture description language \( \Pi \). This forms the basis of our approach and below we sketch various tools and the infrastructure we are currently constructing.

Here we consider mainly an approach to represent the software architecture in such a setting and describe at some level the infrastructure we put up to support the collaborative design process in the sense that the modification and analysis of the architecture is supported.

Related research wrt. development distributed software systems is done in many places. We concentrate here on related approaches which emphasize the architecture design but also provide relations to the implementation stage (e.g., automated application generation or performance prediction). A prominent example is ROI (Regis Orb Implementation, [3, 14]), an integration of the architecture description language REGIS/DARWIN [15] with IONA Technologies CORBA implementation ORBIX. In [13] it is shown how dynamic change can be realized in Darwin architectures. In contrast to Regis/Darwin, \( \Pi \) has a richer language for describing semantic properties in interfaces and also comprises constructs for including performance-related properties.

UniCon (Language for Universal Connector Support, [17, 1]) investigates the idea that within an architecture description, distribution information is attached to connectors between distributed software components. However, our concept for evaluating the architecture’s non-functional properties is more flexible wrt. verification of temporal requirements. In addition, our architecture development environment provides support for generating a distributed application implementation.

Below we now present the \( \Pi \)-way to represent software architecture (cf. chapter 3) followed by a description of

This work is partly funded by the DFG project QUAFOS, contract MU1158/2-1.
checking capabilities within such a representation (cf. chapter 4). The various related tools available or under construction are then briefly described in chapter 5. Finally some concluding remarks regarding our approach and further work close this contribution.

3. ISSUES IN REPRESENTING THE SOFTWARE ARCHITECTURE OF A SYSTEM

Our architecture description language Π supports the design of distributed component-oriented software systems. It provides concepts for separating the development of distributed independent software components from the interconnection and configuration of such components: although component dependency requirements can be stated with a single independent component, the explicit connection structure has to be defined at a different point in the design process. According to [11] a software component is a unit which provides its clients with services specified by its interface and encapsulates local structures that implement these services. Furthermore, it may use services of other components to realize the exported ones. Each component encapsulates one or more abstract data types (ADTs), hence an object-based structuring of the whole architecture is enforced. Collections of components connected via use relations are called configurations. Also, configurations have in principle the same interfaces to their environment as single components and thus may be used as components hierarchically.

In Π each component is described by four sections (cf. figure 1). The export section gives an abstract image of the component’s realization; the abstract data types stated here are public and may be used by other components. The body section describes the realization of a component; here, the construction of the exported abstract data types is encapsulated. According to the concept of formal import, only requirements to imported abstract data types are specified in the import section. While configurations of components are built, the import section has to be actualized with export sections of potential server components via use relations. Finally, in the common parameters section abstract data types are stated which are imported and exported unchanged.

![Component Concept in Π](image)

**Figure 1.** Component Concept in Π

Π is a multi-formalism language and single views can be seen as overlapping partial specifications of a component. Each section can be specified by four views. The type view describes the component’s invariant properties (according to execution of operations) by means of algebraic specification techniques. The imperative view defines imperative operation signatures and algorithms, and the concurrency view specifies possible orderings of operation executions. Finally, the interaction view encapsulates information according to distribution of components.

Due to the fact that in Π each component specification is parameterized by its formal import it can be used with different parameter actualizations. Thus a component developed and viewed in isolation is some kind of component template in contrast to the same component used within the specific context of the other components. Different instantiations of a component which can be connected via use relations are called component incarnations and the isolated component template is called Concurrently Executable Module (CEM). Thus, our approach takes an open world perspective: according to the concept of formal import a clear distinction can be made between the independent development of self-contained CEMs and the connection of component incarnations.

4. CHECKING ARCHITECTURAL PROPERTIES

While in the last chapter we have discussed how the architecture design of distributed and component-oriented software system can be described with the help of the ADL Π, we now focus on how architecture properties can be checked wrt. functional correctness, performance behaviour, and semantics-directed system configuration.

In addition to syntactic correctness of an architecture design specified in Π various classes of context conditions have to be checked (e.g., context conditions guaranteeing consistency wrt. the interplay of different views, compare [12]). A functional correct architecture description in Π can then be used as a foundation to generate a distributed implementation. We have chosen CORBA [16] as a middleware platform for the implementation of the design, because component models in Π and CORBA resemble closely and can easily be integrated and CORBA represents a standard for the interaction of distributed objects. Thus the first step is a mapping from Π to the Object Management Group’s Interface Definition Language IDL [16]. Then our underlying object management system realized in C++ [12] guarantees that all object semantics of the Π design are preserved in the distributed implementation. This is especially important to realize a component’s concurrency properties specified in the design architecture also in the implementation. Details regarding our Π/CORBA integration concept - including the CORBA representation of complex component configurations, the C++ representation of a component’s body section, and the object management system - can be found in [9, 12].

In addition to functional requirements, also non-functional requirements (e.g., response time, throughput, etc.) have essential impact on the design of distributed systems. This is true a priori, i.e. the analysis and assessment of a components’ performance should be possible while the entire design architecture is still unfinished, as well as a
posteriori, i.e. measuring the efficiency of the components’ implementations. Using the Π language, the functional behaviour of distributed components and their connections can be described as well as performance-related attributes of this architecture. For functional as well as performance-related evaluation, we use the Queuing Specification and Description Language QSDL [2, 5]. The transformation of a QSDL-specification to an executable program for simulation and validation of the specified system is performed automatically by the tool QUEST that has been developed at the University of Essen. By executing the simulator, stochastic performance measures can be gained. In [10, 6] we have identified interfaces between the component model in Π and the system specification in QSDL using ViewPoints [7], a method engineering and integration framework. Thus performance requirements of a software system identified in its component model can be evaluated in its corresponding QSDL-system. Finally, the simulation results can be transferred back to the Π world also by means of the ViewPoints framework.

Especially for distributed and component-oriented software systems a mechanism for performing dynamic extensions is important. It is obvious, however, that such a mechanism needs to be based on semantic properties since otherwise non compliant extensions or modifications may lead to disastrous results. Thus it will not suffice to identify distributed and dynamically migrating components only by syntactic and pseudo-semantic means in terms of operation signatures and keyword list trader services. Our approach considers how such semantic properties can be expressed using graphs and how such graphs can be used with the help of graph transformation techniques to express and handle various degrees of compliance between client and server components in distributed environments. The final aim is to enable semantics-directed configuration of distributed and component-oriented software systems. Our approach for semantics-directed component interaction is described in [8] while system configuration by distributed graph transformation is presented in [18].

5. INFRASTRUCTURE AND TOOLS
We devised our tool support strategy for a distributed Π development environment to suit the requirements imposed by the characteristics of distributed component-oriented software systems. Thus the separation of independent component development from component interconnection and configuration demands local tool support for component development and - as here the interplay between different distributed components is involved - server-based tool support for the architectural check and evaluation functionalities presented in chapter 4. Further, dynamically added components have to be identified and accessed in order to suit the demands imposed by the open-world-perspective.

The PI Editing System PIES supports the development of local single CEMs and local configurations. The PIES Java application can be downloaded via the WWW site of the University of Essen’s computer science department (www.informatik.uni-essen.de/Fachgebiete/SoftTech). In addition to local component development the PIES application offers remote connections via Java’s communication mechanism Remote Method Invocation RMI to the tools CorrCheck and CodeGen implementing functional correctness checking and implementation generation based on CORBA. Ongoing work is the integration of the tools QUEST (developed at the University of Essen by B. Müller-Clostermann et al. [4]) for performance evaluation and AGG (developed at the Technical University of Berlin by G. Taentzer [19]) for manipulation of component semantics graphs.

While this tool infrastructure requires to download and execute the PIES Java application, our final aim is to provide the entire functionality within a PIES Java applet which can be accessed from any WWW browser (cf. figure 2).

![Figure 2. Distributed PIES Development Environment](image-url)

6. CONCLUSIONS AND FURTHER WORK
In this contribution we have sketched a WWW-based development environment for distributed software architectures. Firstly, we have introduced the ADL Π as a set of formalisms to support independent component design as well as collaboration and configuration of distributed components. Then we have discussed how architectural properties can be checked and used in order to generate an application implementation, predict the architecture’s performance behaviour, and locate distributed components with the help of a graph-based semantics description. Finally, we have presented a distributed and WWW-based development environment as tool support for our approach.
While a RMI-based tool prototype including component and configuration development, functional correctness checking, and application implementation generation on top of CORBA is already completed, we are working on integrating QUEST and AGG as well as on supporting distributed user data repositories. Also, issues like security and repository access rights need to be investigated.

7. ACKNOWLEDGEMENTS

The PI-language was developed while the first author was with H. Weber, now head of the Fraunhofer Institute ISST Berlin/Dortmund, Germany. His inspiration is gratefully acknowledged. The ViewPoint framework was devised while the first author was with A. Finkelstein, J. Kramer, B. Nuseibeh, and L. Finkelstein. Their support is thankfully appreciated. We would like to thank B. Sucrow and G. Taentzer for their helpful inspirations regarding graph transformation and distributed graph transformation and B. Müller-Clostermann for many fruitful discussions regarding performance evaluation.

8. REFERENCES


